Name : VEDASHREE BHALERAO

Class : LY – AIA – 1

Batch : B

Roll No : 2213191

Assignment No. 7

Aim: Develop RNN model for Cryptocurrency pricing prediction or text sentiment analysis

Objectives:

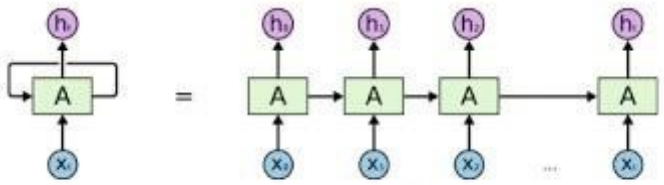
1. To learn RNN 2. To learn and implement LSTM

Theory:

RNN

A **recurrent neural network** (**RNN**) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Derived from feedforward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition[4] or speech recognition.

The term “recurrent neural network” is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is finite impulse and the other is infinite impulse. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that cannot be unrolled.



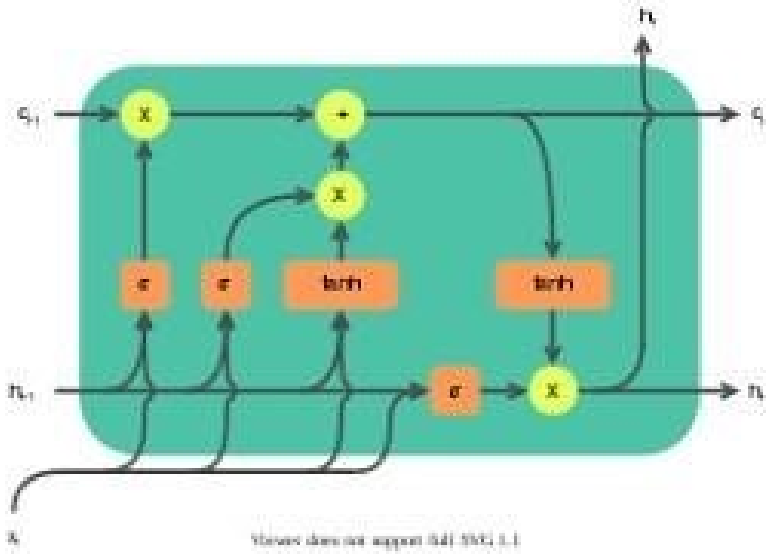
An unrolled recurrent neural network.

LSTM

Long short-term memory (**LSTM**) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can not only process single data points (such as images), but also entire sequences of data (such as speech or video). For example, LSTM is applicable to tasks such as unsegmented, connected handwriting recognition, speech recognition and anomaly detection in network traffic or IDSs (intrusion detection systems).

A common LSTM unit is composed of a **cell**, an **input gate**, an **output gate** and a **forget gate**. The cell remembers values over arbitrary time intervals and the three *gates* regulate the flow of information into and out of the cell.

LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series.
LSTMs were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs. Relative insensitivity to gap length is an advantage of LSTM over RNNs, hidden Markov models and other sequence learning methods in numerous applications.



BPTT Backpropagation refers to

two things:

- The mathematical method used to calculate derivatives and an application of the derivative chain rule.
- The training algorithm for updating network weights to minimize error. It is this latter understanding of backpropagation that we are using here.

The goal of the backpropagation training algorithm is to modify the weights of a neural network in order to minimize the error of the network outputs compared to some expected output in response to corresponding inputs.

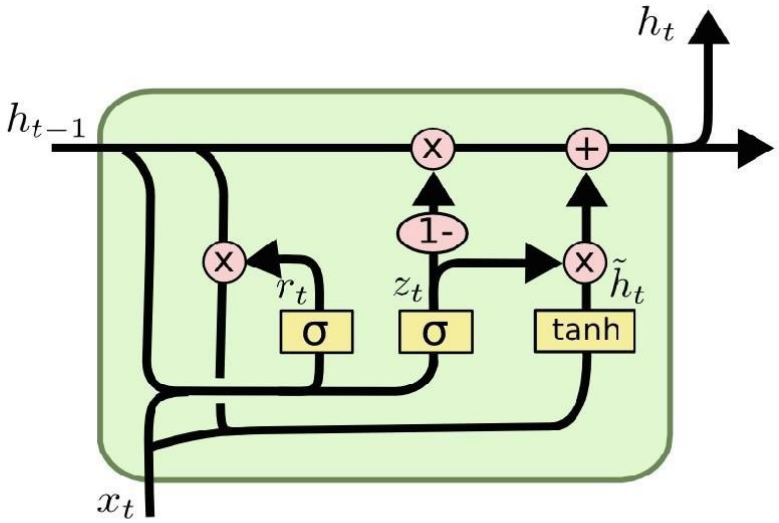
It is a supervised learning algorithm that allows the network to be corrected with regard to the specific errors made.

The general algorithm is as follows:

1. Present a training input pattern and propagate it through the network to get an output.
2. Compare the predicted outputs to the expected outputs and calculate the error.
3. Calculate the derivatives of the error with respect to the network weights.
4. Adjust the weights to minimize the error.
5. Repeat.

GRU

Gated recurrent units (**GRU**s) are a gating mechanism in recurrent neural networks, introduced in 2014 by Kyunghyun Cho et al. The GRU is like a long short-term memory (LSTM) with a forget gate, but has fewer parameters than LSTM, as it lacks an output gate. GRU's performance on certain tasks of polyphonic music modeling, speech signal modeling and natural language processing was found to be similar to that of LSTM. GRUs have been shown to exhibit better performance on certain smaller and less frequent datasets.



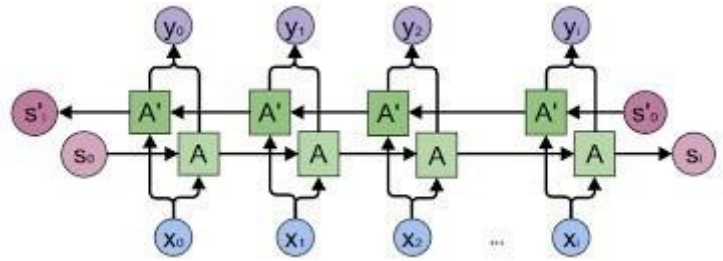
Bi directional RNN

Bidirectional recurrent neural networks(RNN) are really just putting two independent RNNs together. The input sequence is fed in normal time order for one network, and in reverse time order for another. The outputs of the two networks are usually concatenated at each time step, though there are other options, e.g. summation.

This structure allows the networks to have both backward and forward information about the sequence at every time step. The concept seems easy enough. But when it comes to actually implementing a neural network which utilizes bidirectional structure, confusion arises…

The first confusion is about **the way to forward the outputs of a bidirectional RNN to a dense neural network**.

The second confusion is about the **returned hidden states**. In seq2seq models, we'll want hidden states from the encoder to initialize the hidden states of the decoder. Intuitively, if we can only choose hidden states at one time step(as in PyTorch), we'd want the one at which the RNN just consumed the last input in the sequence. But **if** the hidden states of time step *n* (the last one) are returned, as before, we'll have the hidden states of the reversed RNN with only one step of inputs seen.



Code:

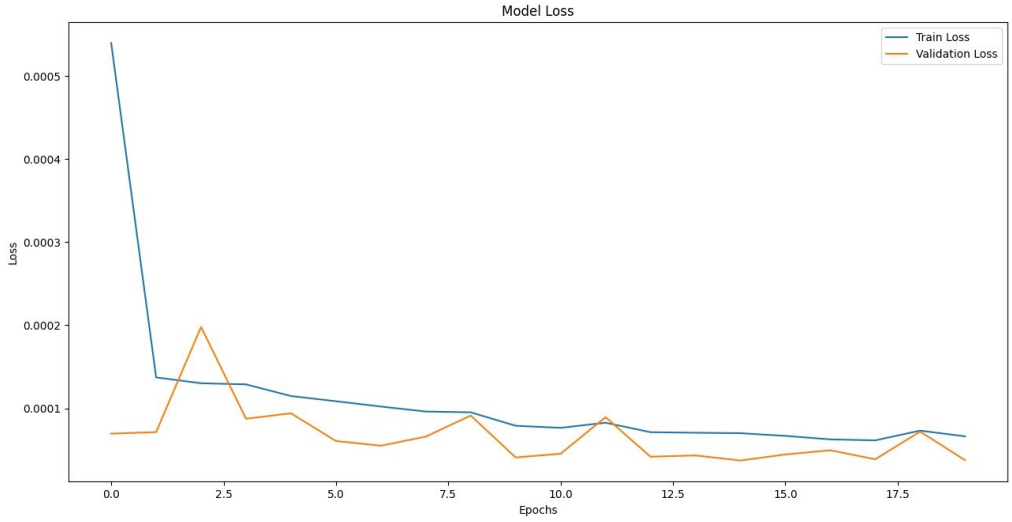
```
# Importing necessary libraries
import numpy as np import
pandas as pd import
matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler from
tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, GRU, Bidirectional,
Dropout

# Load dataset
df = pd.read_csv("Assets/BTC_USD.csv", parse_dates=['date']) df
= df.sort_values('date')
df.reset_index(drop=True, inplace=True)

# Data Preprocessing
# Using MinMaxScaler to scale data between 0 and 1
scaler = MinMaxScaler() close_price =
df[['close']].values
scaled_close = scaler.fit_transform(close_price)

# Define sequence length and prepare data sequences for RNN
SEQ_LEN = 60  # sequence length (60 days) def
to_sequences(data, seq_len=SEQ_LEN):     X, y = [], []
for i in range(len(data) - seq_len):
X.append(data[i:i + seq_len])
        y.append(data[i + seq_len])
return np.array(X), np.array(y)

X, y = to_sequences(scaled_close)
```

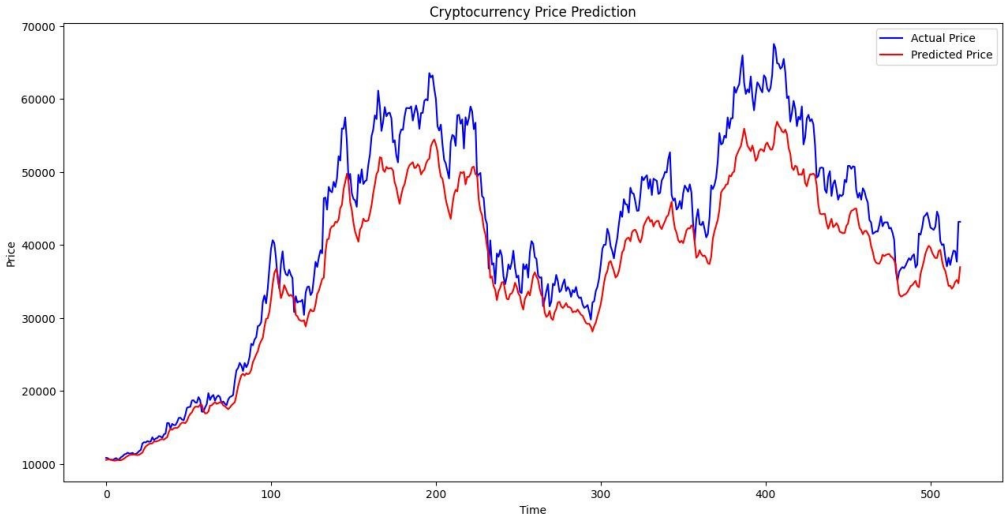
```
# Split data into training and testing sets (80% train, 20% test)
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
# Model Architecture model = Sequential()

# Adding Bidirectional LSTM layer
model.add(Bidirectional(LSTM(64, return_sequences=True),
input_shape=(X_train.shape[1], 1))) model.add(Dropout(0.2))

# Adding GRU layer
model.add(GRU(32, return_sequences=False))
model.add(Dropout(0.2))

# Dense layer for output model.add(Dense(1))

# Compiling the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Training the model
history = model.fit(X_train, y_train, epochs=20, batch_size=32,
validation_split=0.1)

# Model evaluation
model.evaluate(X_test, y_test)

# Plotting training and validation loss
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend() plt.show()

# Making Predictions predictions =
model.predict(X_test)
predicted_price = scaler.inverse_transform(predictions) actual_price
= scaler.inverse_transform(y_test)

# Plotting Actual vs Predicted Prices
plt.plot(actual_price, color='blue', label='Actual Price')
plt.plot(predicted_price, color='red', label='Predicted Price')
plt.title('Cryptocurrency Price Prediction') plt.xlabel('Time')
plt.ylabel('Price') plt.legend() plt.show()
```

Results:

```
Epoch 1/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 7s 42ms/step - loss: 0.0013 - val_loss: 6.9518e-05
Epoch 2/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - loss: 1.5053e-04 - val_loss: 7.1398e-05
Epoch 3/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 3s 42ms/step - loss: 1.2944e-04 - val_loss: 1.9792e-04
Epoch 4/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - loss: 1.4095e-04 - val_loss: 8.7458e-05
Epoch 5/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 2s 36ms/step - loss: 1.0291e-04 - val_loss: 9.4002e-05
```

```
Epoch 16/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 2s 38ms/step - loss: 6.7753e-05 - val_loss: 4.4484e-05
Epoch 17/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 2s 35ms/step - loss: 6.9390e-05 - val_loss: 4.9524e-05
Epoch 18/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - loss: 5.9826e-05 - val_loss: 3.8700e-05
Epoch 19/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - loss: 7.6771e-05 - val_loss: 7.2000e-05
Epoch 20/20
59/59 ━━━━━━━━━━━━━━━━━━━━ 2s 33ms/step - loss: 6.3870e-05 - val_loss: 3.7543e-05
17/17 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0043
```





Conclusion:

Thus, we have learned how to code LSTMs and understood How Recurrent neural networks work